

Debug

Philippe PRADOS

philippe@prados.net



*Préservez l'environnement,
n'imprimez pas ce document*

TABLE DES MATIERES

1.	Introduction.....	3
2.	La librairie.....	4
2.1	Déclaration des paquets de classes.....	4
2.2	Les traces.....	5
2.3	Exemple.....	6
2.4	D'autres outils.....	7
3.	Environnement.....	7

Avant propos

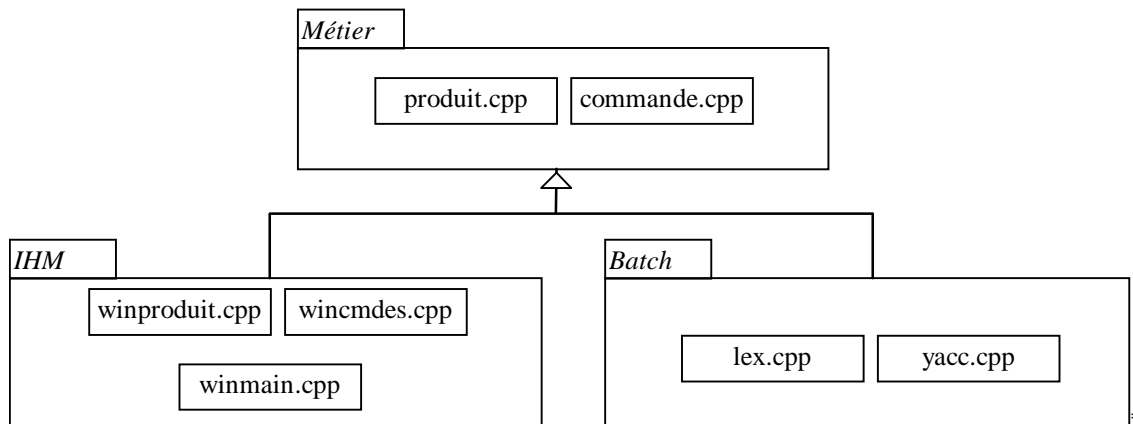
Ce document propose une API objet de trace de source C++.

1. INTRODUCTION

Pour pouvoir tracer l'exécution d'un programme, il faut utiliser des outils logiciels spécialisés. Ce document explique la librairie proposée. Celle-ci permet de sélectionner les éléments à tracer.

Chaque fichier peut être tracé individuellement. Il peut également appartenir à un groupe de fichiers (Files Package). Il est alors possible de tracer l'ensemble des fichiers appartenant à ce groupe. De plus, un groupe peut hériter d'un ou de plusieurs autres groupes de fichiers. Il est alors possible de tracer une hiérarchie complète de groupe de fichiers.

Prenons un exemple. Les classes d'un *métier* sont séparées dans deux fichiers sources : `produit.cpp` et `commande.cpp`. L'interface Homme-Machine (IHM) est décrite dans trois fichiers : `winproduit.cpp`, `wincmdes.cpp` et `winmain.cpp`. Un langage auteur de traitement par lot (batch) et décrit dans deux fichiers : `lex.cpp` et `yacc.cpp`.



L'IHM hérite du *Métier*. L'ensemble *Batch* hérite également du *Métier*.

Lors du débogage du programme, le développeur désire par exemple, afficher les traces du fichier `winproduit.cpp`. Si les informations ne sont pas suffisantes, il peut désire afficher l'ensemble des traces concernant l'IHM. Avec cette option, les fichiers `winproduit.cpp`, `wincmdes.cpp` et `winmain.cpp` seront tracés. Peut-être que cela n'est toujours pas suffisant, dans ce cas, il est possible de demander la trace du *Métier* et de tous les paquets de fichiers dérivés. Il est possible d'exclure certains paquet de fichiers ou un fichier particulier de la trace. Toutes ces modifications se font sans recompilation !

Pour gérer cela, chaque fichier peut déclarer appartenir à un paquet de fichiers. Cette déclaration est optionnelle. Un fichier peut n'appartenir à aucun paquet particulier.

Un fichier possède un paramètre pouvant avoir trois valeurs possibles : `Unknow`, `Off` et `On`. La valeur `Unknow` indique que la validation de la trace pour ce fichier n'est pas déclarée. C'est la valeur par défaut de tous les fichiers. La trace sera effectuée ou non suivant l'état du paquet de fichiers correspondant. Les valeurs `On` et `Off` indiquent l'état de la trace pour ce fichier. Ces valeurs sont prioritaires sur la valeur du paramètre du paquet de fichiers associé.

De même, un paquet de fichiers possède un paramètre ayant trois valeurs possibles. `Unknow` indique que l'état de trace de ce paquet n'est pas connu. Le paramètre global résoudra cela (voir plus bas). Les valeurs `On` et `Off` indiquent l'état de trace du paquet. Tous les fichiers de ce paquet ayant l'état `Unknow` sont valorisés de même.

Un paquet de fichiers peut hériter d'un ou de plusieurs autres paquets. Dans ce cas, si au moins un des paquets de base indique la valeur `On`, ce paquet est à `On`. Sinon, si au moins un des paquets est à `Off`, ce paquet est à `Off`. Sinon, l'état est `Unknow`.

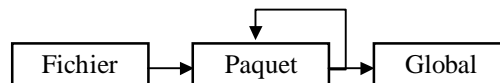
Il existe un paramètre global de validation du mécanisme de trace. Ce paramètre permet d'offrir ou de supprimer l'ensemble des traces. Ce paramètre peut avoir trois valeurs possibles : `Off`, `All` et `On`. `Off` indique l'absence totale de trace. `All` indique que tous les fichiers n'étant

pas déclarés explicitement comme *Off* sont à *On*. La valeur *On* pour ce paramètre indique que tout fichier explicitement déclarés a *On* est tracé. Les autres ne le sont pas.

Voici la table de vérité de ces trois paramètres.

Global	Paquet	Fichier	État
Off	/	/	Off
All	Unknown	Unknown	On
All	Unknown	Off	Off
All	Unknown	On	On
All	Off	Unknown	On
All	Off	Off	Off
All	Off	On	On
All	On	Unknown	On
All	On	Off	Off
All	On	On	On
On	Unknown	Unknown	Off
On	Unknown	Off	Off
On	Unknown	On	On
On	Off	Unknown	Off
On	Off	Off	Off
On	Off	On	On
On	On	Unknown	On
On	On	Off	Off
On	On	On	On

Pour simplifier, lorsque le paramètre global est à *On*, les paramètres sont consultés via une " chaîne de responsabilité "



- Si l'attribut du fichier est *Unknow*, l'algorithme délègue la récupération de la valeur au paquet correspondant.
- Si l'attribut du paquet est *Unknow*, l'algorithme délègue la récupération de la valeur aux paquets hérités. S'il n'existe aucun paquet hérité, il délègue la récupération au paramètre global.

Finalement, cela est intuitif. Le comportement est en général celui que l'on imagine.

2. LA LIBRAIRIE

Pour bénéficier des services de trace, il faut inclure le fichier "*debug.h*". Celui-ci inclut les fichiers *<assert.h>*, *<stdlib.h>*, *<iostream.h>*, *<ibase.h>* et "*indent.hh*".

2.1 Déclaration des paquets de classes

Il faut maintenant regarder comment déclarer l'association entre un fichier et le paquet correspondant. Pour cela, il faut déclarer une variable globale de type `CDBG::CDeclareFilePackage`. Le constructeur de cette classe demande deux chaînes de caractères. La première indique le nom du fichier ; la deuxième le nom du paquet de classe.

```
static CDBG::CDeclareFilePackage dclFilePackage(__FILE__, "Metier");
```

Seul le nom du fichier est pris en compte. Le répertoire ou l'extension éventuelle ne sont pas mémorisés. Cela permet de considérer les fichiers en-tête de même nom, comme ayant le même comportement vis-à-vis de la trace que le fichier "*.cpp*". Par exemple, les fichiers "*produit.cpp*" et "*produit.h*" sont vus comme les mêmes. Ils ont le même comportement vis-à-vis de la trace.

Pour déclarer un héritage entre paquets de fichiers, il faut de même instancier une variable locale du type `CDeclareDerivedPackage`. Le constructeur de cette classe attend deux chaînes de caractères. La première indique le paquet de classe dérivé, et la deuxième le paquet de classe dont il hérite. Il peut y avoir plusieurs héritages pour le même paquet de fichiers. Dans ce cas, il faut déclarer autant de variables globales que de paquets de bases.

```
static CDeclareDerivedPackage _dclDerivedPackage1("Windows", "Metier");
static CDeclareDerivedPackage _dclDerivedPackage2("Windows", "IHM");
```

Ces lignes de déclarations doivent être encadrées par :

```
#ifndef NDEBUG
...
#endif
```

afin de les supprimer lors de la compilation finale du produit. Le `#define NDEBUG` est compatible avec le standard ANSI C. Il permet de supprimer toutes les assertions du programme.

D'autre part, le paramétrage globale des traces est présent dans la variable `CDBG::global`. Celle-ci est déclarée en `volatile` pour pouvoir être modifié par un débogueur.

2.2 Les traces

Pour afficher des informations dans le programme, il faut utiliser une collection de macros. Celles-ci permettent de tester l'application lors de son exécution. L'ensemble des macros est supprimable à l'aide du `#define NDEBUG`. Ce `#define` est le même que pour supprimer les `assert()` conformément à la norme C ANSI. L'affichage de ces macros dépend du fichier où elles sont utilisées. Les règles de validation étant celles décrites ci-dessus. La variable du préprocesseur `__FILE__` est utilisée pour reconnaître le fichier d'utilisation d'une macro.

2.2.1 Flux de debug

Il existe deux flux particuliers pour utiliser les traces.

- `ctrace` est un flux permettant de tracer des informations. Ce flux n'est valide que si la trace pour le fichier où il est utilisé est valide. Ce flux est *branché* sur `cerr`.
- `cpackage(name)` est un flux dont la validité dépend de l'état du paquet `name` indiqué.

Le flux `ctrace` est utilisé dans les macros suivantes.

2.2.2 Les assertions

Il y a trois catégories de macros. La première permet de vérifier le programme.

```
#define ASSERTMSG(tst,msg)
#define PRECONDITION(tst,msg)
#define POSTCONDITION(tst,msg)
#define INVARIANT(tst,msg)
```

Ces macros permettent d'effectuer le même traitement qu'un appel à un `assert()` classique mais avec la possibilité d'ajouter un texte explicatif. La macro `ASSERTMSG` est identique à `assert()` sauf qu'il faut ajouter un message. Celui-ci est du type flux C++. Il est possible d'écrire :

```
ASSERTMSG(pt!=NULL,"Pointeur a null dans l'objet " << *this);
```

Il devient intéressant d'avoir un opérateur de flux pour tous les objets. Ainsi, il est possible de tracer les objets lors d'erreurs.

```
{ A a;
  ASSERTMSG(i<10,a);
}
```

Le message affiché lors d'une assertion invalide est le suivant

```
Assertion failed : tst, msg, file f, line l
```

Le fichier et la ligne où l'`assert` est écrit sont indiqués dans le message. Le traitement est ensuite interrompu par un `abort()`. Cela génère un `core` ce qui permet d'appeler le débogueur pour connaître le contexte de l'erreur.

La macro `PRECONDITION` est identique à la macro `ASSERTMSG` sauf que le message est préfixé de "Pre-condition".

La macro `POSTCONDITION` est identique à la macro `ASSERTMSG` sauf que le message est préfixé de "Post-condition". Il est possible de supprimer de la compilation tous les appels à cette macro en déclarant `#define NOPOST`.

La macro `INVARIANT` est identique à la macro `ASSERTMSG` sauf que le message est préfixé de "Invariant". Il est possible de supprimer de la compilation tous les appels à cette macro en déclarant `#define NOINVARIANT`.

Une fois les tests unitaires faits, les Post-conditions et les Invariants peuvent être considérées comme toujours valides. Je ne recommande pas la suppression de ces tests, sauf si les performances deviennent nettement meilleures et que cela constitue un frein à l'intégration. Il est préférable de garder dans le code le plus longtemps possible les "auto-tests" car ils ont une sémantique forte, qualité qu'un simple "core" n'a pas.

2.2.3 Les passages

La deuxième catégorie de macro permet d'afficher des traces du passage par certains points du programme.

```
#define PASS(msg)
#define PASSINC(msg)
#define PASSDEC(msg)
#define ENTRY(name)
#define EXIT(name)
```

La macro `PASS` permet d'afficher un message du type

```
Pass in file(line), msg
```

La macro `PASSINC` est identique à la macro `PASS` sauf que l'indentation courante du flux `ctrace` est incrémentée¹ de un.

La macro `PASSDEC` est identique à la macro `PASS` sauf que l'indentation courante du flux `ctrace` est décrémentée de un.

La macro `ENTRY` permet d'afficher un message en entrée de fonction du type

```
Pass in file(line), enter name
```

le flux `ctrace` incrémente l'indentation.

La macro `EXIT` permet d'afficher un message en sortie de fonction du type

```
Pass in file(line), exit name
```

le flux `ctrace` décrémente l'indentation.

2.2.4 Les traces

La troisième catégorie de macro permet d'afficher une trace d'une variable. Pour une écriture du type :

```
{ int i=99;
  TRACE(i);
}
```

la trace affiche :

```
i=99
```

Tous les objets pouvant être affichés par un flux peuvent être manipulés par cette macro.

```
{ A a;
  TRACE(a); // a=...
}
```

2.3 Exemple

Voici un exemple d'utilisation de ces macros.

```
void f()
{ ENTRY("f");
  PASS("Je suis dans f");
  EXIT("f");
}

void main()
{ ENTRY("main");
  PASSINC("Debut Etape 1");
  int i=99;
  //...
  TRACE(i);
  PASSDEC("Fin Etape 1");
  PASSINC("Debut Etape 2");
  PRECONDITION(i==99,"i doit etre a 99");
  f();
  f();
}
```

¹ Pour l'indentation des flux voir le livre "La qualité en C++"

```
//...
PASSDEC("Fin Etape 2");
EXIT("main");
}
```

La trace affichée à l'écran est celle-là :

```
Pass in tst.cc(10), enter main
  Pass in tst.cc(11), Debut Etape 1
    i==99
  Pass in tst.cc(13), Fin Etape 1
  Pass in tst.cc(14), Debut Etape 2
    Pass in tst.cc(5), enter f
      Pass in tst.cc(6), je suis dans f
    Pass in tst.cc(7), exit f
  Pass in tst.cc(5), enter f
    Pass in tst.cc(6), je suis dans f
  Pass in tst.cc(7), exit f
  Pass in tst.cc(16), Fin Etape 2
Pass in tst.cc(17), exit main
```

L'indentation permet de visualiser les appels imbriqués.

2.4 D'autres outils

Pour pouvoir vérifier lors de la compilation, de l'héritage d'une classe par une autre, il existe deux outils. La fonction `checkBase(Obj1,Obj2)` retourne `true` si la classe de `Obj1` est héritée par la classe de `Obj2`. La fonction `checkDerived(Obj1,Obj2)` retourne `true` si les classes de `Obj1` et `Obj2` possèdent une relation d'héritage (`checkBase(Obj1,Obj2) || checkBase(Obj2,Obj1)`). Ces fonctions sont `inline` et ne sont compilées que par les valeurs `true` ou `false`.

Sous Unix, les fonctions `isHeap()` et `isData()` permettent de vérifier la localisation d'un pointeur. `isHeap(pt)` retourne `true` si `pt` représente une adresse du tas. `isData(pt)` retourne `true` si `pt` représente l'adresse d'une variable globale. De plus, une capture des signaux permet de générer un `core` pour pouvoir analyser la pile.

3. ENVIRONNEMENT

Le paramètre global des traces est la variable d'environnement `DEBUG`. Celle-ci peut prendre les valeurs : `On`, `Off`, `all`, `O`, `N`, `A`, `1` ou `0`.

Pour indiquer l'état de trace pour chacun des fichiers et des paquets de fichiers, il faut rédiger un fichier "`debug.ini`". Celui-ci utilise une syntaxe qui se traduit dans le format BNF comme cela :

<i>blocs</i>	:	<i>files</i> <i>packages</i> <i>blocs</i>		
<i>files</i>	:	[Files]\n		<i>list-files</i>
<i>list-files</i>	:	<i>filename</i> <i>list-files</i>	=	<i>yes-no</i>
<i>packages</i>	:	[Packages]\n		<i>list-packages</i>
<i>list-packages</i>	:	<i>filename</i> <i>list-packages*</i>	=	<i>yes-no</i>
<i>yes-no</i>	:	<i>on</i> <i>yes</i> <i>no n off 0</i>		<i>off</i> <i>1</i>
			<i>y</i>	<i>on</i>

Voici un exemple de cette syntaxe :

```
[Packages]
Metier=On
Batch=Off

[Files]
winmain=Off
```

Il peut y avoir plusieurs sections `[Packages]` ou `[Files]`.